FAME: FAirly MEasuring Multithreaded Architectures

Javier Vera[1] Francisco J. Cazorla[1] Alex Pajuelo[2]
Oliverio J. Santana[3] Enrique Fernández[3] Mateo Valero[1,2]

[1]Barcelona Supercomputing Center, Spain. {javier.vera, francisco.cazorla}@bsc.es
[2]DAC, Universitat Politècnica de Catalunya, Spain. {mpajuelo, mateo}@ac.upc.edu.
[3]Universidad de Las Palmas de Gran Canaria, Spain. {ojsantana, efernandez}@dis.ulpgc.es

Abstract

Nowadays, multithreaded architectures are becoming more and more popular. In order to evaluate their behavior, several methodologies and metrics have been proposed. A methodology defines when the measurements of a given workload execution are taken. A metric combines those measurements to obtain a final evaluation result. However, since current evaluation methodologies do not provide representative measurements for these metrics, the analysis and evaluation of novel ideas could be either unfair or misleading. Given the potential impact of multithreaded architectures on current and future processor designs, it is crucial to develop an accurate evaluation methodology for them.

This paper presents FAME, a new evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME reexecutes all traces in a multithreaded workload until all of them are fairly represented in the final measurements taken from the workload. We compare FAME with previously used methodologies for both architectural research simulators and real processors. Our results show that FAME provides more accurate measurements than other methodologies, becoming an ideal evaluation methodology to analyze proposals for multithreaded architectures.

1 Introduction

Thread-level parallelism is a common strategy for improving processor performance. Since it is difficult to extract more instruction-level parallelism from a single program, multithreaded processors rely on using the additional transistors to obtain more parallelism by simultaneously executing several tasks. This strategy has led to a wide range of multithreaded processor architectures, including simultaneous-multithreaded processors (SMT) [14][15][21], chip multiprocessors (CMP) and also CMP/SMT processors, *i.e.*, chip multiprocessors in which every core is a SMT [18].

To design these processors, the first steps commonly involve using simulation tools [7][21] to model their expected behavior. These simulators allow researchers to propose and test novel techniques that could be included in the final processor design. In order to evaluate these new techniques, computer architecture researchers use benchmark suites [1][2], since they are representative of current and future applications that will be executed by the designed processor. In spite of the increasing trend to use truly parallel applications, they are currently less common in real machines than non-cooperative single-threaded applications. Therefore, computer architecture researchers frequently evaluate multithreaded processors using workloads composed by non-cooperative single-threaded applications, picked up from a benchmark suite, which perform non-related work and do not communicate each other.

However, as the complexity of the simulated processor grows, the simulator also becomes more complex, increasing the time required for completing benchmark simulations. As a consequence, the amount of time required to simulate a whole benchmark becomes unaffordable. The most common approach to reduce simulation time is to select a smaller segment of every benchmark that is representative of the whole execution [9][11][17][25]. This representative segment (from now onwards we will call it *trace*) will be used to feed the simulator with the data required to evaluate the processor model.

The generation of representative traces allows to reduce simulation time in traditional single-threaded processors. Nevertheless, using those single-thread traces in multithreaded processors is not straightforward. Working with several traces at a time involves an important decision, that is, to determine when a simulation finish. In a single-threaded processor, the simulator runs the full trace until completion. However, it is not so easy in a multithreaded processor simulator running a workload composed by several traces. Traces

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE

in a workload can execute at different speeds due to the different features of each program, as well as the availability of the shared resources. Therefore, they do not have to necessarily complete execution at the same time. We will explain this fact with an example. Let us assume a M-context multithreaded processor executing a 2-thread workload (being M greater than or equal to 2). The execution of this workload occurs as depicted in Figure 1. Both threads execute at different speeds and thus they do not have to finish at the same time. Therefore, we can divide the execution of the workload into two phases. Firstly, a *multithreaded period* in which both threads are being executed. Secondly, after the first thread finishes (Thread 0 in Figure 1), there is a *single-threaded period* in which the remaining thread executes alone until completion. If the multithreaded period is too short, then the potential of the multithreaded processor is only exploited during a small interval of time. As a consequence, the *total execution time* becomes an inaccurate metric for multithreaded processors. We found that, when executing 2-thread workloads composed by SPEC benchmarks in our simulated processor, a 2-context SMT simulator spends almost one third of the time executing a single thread.

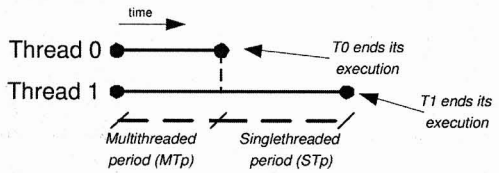


Figure 1. Execution of a 2-thread workload in a M-context multithreaded processor ($M \geq 2$).

Generally, the execution of an N-thread workload [1] involves N periods of N, N-1,... and 1 thread respectively. A common characteristic of all methodologies we have analyzed is that, only measures obtained from the period with N running threads are representative. Periods with less running threads should not be taken into account since the results could be inaccurate.

In this paper, we analyze several simulation methodologies that have been used to face this problem. These methodologies suggest how simulation should be performed and, in particular, they determine when workload simulations have to finish. However, we show that these methodologies cannot ensure that the trace of every benchmark is fully executed, and thus it is not possible to assure that the measurements obtained are representative of the whole program behavior.

To face this problem, we present FAME, a new simulation methodology for the evaluation of multithreaded processors. FAME can be used with any of the state-of-the-art tools for obtaining a representative trace of a given program [9][11][16][17][25] and can be applied to any multithreaded architecture like SMT, CMP or SMT/CMP. We present results for both a well-known SMT simulation tool (SMTsim) and a real SMT processor (Intel Pentium 4). Nevertheless, there is no loss of generality. FAME is applicable to any multithreaded design, since all them present identical evaluation problems. Overall, our results show that FAME provides more accurate measurements than previously used methodologies.

[1] We assume that the number of threads in a workload is smaller than or equal to the number of available hardware contexts in a multithreaded processor.

2 Experimental Environment

This section describes the research scenario we use to compare existing evaluation methodologies and FAME. As mentioned above, FAME can be applied to both simulation environments and real processors. We use a different experimental methodology in each environment.

2.1 Simulation Environment

To evaluate FAME in a state-of-the-art experimental environment, we use an SMT simulator derived from smtsim [21] (see configuration parameters in Table 1). As an illustration of applicability of the FAME methodology we have selected two well-known fetch policies: *icount* [21] and *stall* [20]. The icount fetch policy prioritizes those threads with fewer instructions in the processor pipeline. The stall fetch policy uses the same heuristic, but it also detects whenever a thread has a pending long-latency memory access. When this situation is detected, stall prevents the thread from fetching more instructions until the memory access is resolved, avoiding unnecessary over-pressure over the shared resources.

Table 1. Baseline configuration.

Parameter	Value
Pipeline depth	12 stages
Number of contexts	2 and 4
Default fetch policy	icount
Fetch/Issue/Commit Width	8
Queue Entries	80 int, 80 fp, 80 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	320 integer, 320 fp
(shared)ROB size	512 entries
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way assoc.
Return Address Stack	256 entries
Icache, Dcache	64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	2048 Kbytes, 8-way, 8-bank, 64-byte lines, 20 cycle access
Main memory latency	300 cycles
TLB miss penalty	160 cycles

We feed our simulator with traces collected from the whole SPEC2000 benchmark suite [2] (excluding *facerec*, *fma3d* and *sixtrack*, from which we were unable to collect traces) using the reference input set. Benchmarks were compiled with the Compaq/Alpha C V5.8-015 compiler on Compaq UNIX V4.0 with all the optimizations enabled. Each trace contains 300 million instructions, which were selected using SimPoint [16] to analyze the distribution of basic blocks. Using these benchmarks, we generated workloads

0-7695-2944-5/07 $25.00

with all possible 2-thread combinations, leading to a total number of 276 workloads [2].

2.2 Real Processor Environment

To evaluate FAME in a real processor we use an Intel Pentium 4 3Ghz processor with Hyperthreading Technology and 512 MBytes of DDRAM at 400 Mhz. The operating system is a Fedora Core 3 with gnu linux kernel 2.6.11 patched with perfctr-2.6.18 to allow the access to the performance monitoring counters from any privilege level of execution. The operating system is booted at runlevel 1 to reduce as much as possible the interferences generated by multiuser-multitasking processing. Video, audio and communication hardware capabilities are disabled. Gcc 3.4.2 and the Intel Fortran Compiler 9.0 were used to compile the whole SPEC2000 benchmark suite with all optimizations enabled. Benchmarks are executed until completion with the reference input set. As in the simulation environment, the SMT workloads were generated with all combinations of 2 applications from SPEC2K. In this case we use the whole benchmark suite, leading to 351 2-thread combinations.

3 Evaluating Multithreaded Processors

Measuring the performance of multithreaded processors is a complex task. Several *methodologies* and *metrics* have been proposed in the literature. A methodology defines how simulation is performed and when the measurements are taken. Later, a metric combines those measurements to obtain a final result of the performance of the evaluated processor. Current metrics to measure the performance of multithreaded processor include the IPC Throughput, the Weighted Speedup [19] and the Harmonic Mean [13]. The final result for a given workload is based on both the IPC achieved by each thread in a workload and the IPC of each thread when it is run in isolation. All of these metrics are based on per-thread IPC and it can be proven mathematically that the maximal error we obtain with either Weighted Speedup, throughput or harmonic mean for a given workload is lower than or equal to the maximal error incurred in measuring the workload per-thread IPC. Therefore, in this paper we will show all our results in terms of per-thread IPC, since all other metrics we could obtain will have a lower error than the error of per-thread IPC.

Two main parameters define the behavior of a simulation methodology, the *trace duration* and the *finalization moment*.

Trace duration: Researchers frequently use the SimPoint tool [16] to select a representative trace of S instructions from the whole program. We differentiate two kinds of traces, *fixed length traces* and *variable length traces*. If we use a fixed length trace and, when running a multithreaded simulation, it is required to execute more than S instructions, the trace is re-executed from the beginning. If we use a variable length trace schema, instructions beyond the trace of S instructions are executed as needed until the workload simulation ends. The first drawback of the latter strategy is that it is not possible to know beforehand the total number of instructions to execute beyond S, since it depends on the processor setup and the other threads in the workload. Therefore, an accurate upper bound of the number of required instructions cannot be obtained. A second drawback is that there is no guarantee that the instructions after the interval provided by SimPoint are representative of the program. Due to these two drawbacks, we use fixed length traces in our study, which is according to the SimPoint philosophy.

[2]Note that, if a workload is composed by benchmarks A and B, the workload with benchmarks B and A is not generated.

Table 2. A possible classification of current methodologies.

Finaliz. Moment → / Trace duration ↓	First	Last	Fixed Instructions		
			100 mill	200 mill	1 bill
Fixed Length	[8]				
Variable Length	X	[26]	[23]	[12]	[10]

Finalization moment: In order to fairly evaluate the performance of an SMT processor, measurements should be obtained while all threads in a given workload are running. However, the threads in a workload can be executed at different speeds, and thus they do not have to finish at the same time. Consequently, the evaluation methodology should determine what to do whenever any thread finalizes its execution. All current simulation methodologies can be classified based on the finalization moment. This classification, shown in Table 2, includes the *First*, *Last*, and *Fixed Instructions* methodologies.

3.1 Current Evaluation Methodologies

The First methodology finalizes the simulation of a workload when any thread of the workload ends its execution [8]. The main drawback of this methodology is that only one trace in the workload is executed until completion, and thus it cannot be ensured that the remaining traces execute completely, losing representativity in the final result.

The Last methodology finalizes workload simulation when all the traces have been run until completion. When any trace ends, excluding the last one, it can either reexecute (fixed length traces) or continue execution beyond that point (variable length traces) [26] while the other traces are still executing. The main drawback of this methodology is that the total number of evaluated instructions can vary from an evaluation to another one. Since the execution speed of the different threads depends on the processor parameters, any variation can cause all threads to be executed at different speeds. As a consequence, it cannot be ensured that the amount of executed instructions is the same for different simulations with different parameter values, and thus comparisons between them may be inaccurate.

The Fixed Instructions methodology is based on the idea of executing the same amount of instructions in every sim-

ulation. The simulation finalizes whenever the total number of executed instructions reaches a fixed threshold. This threshold is usually determined per thread, that is, the simulation of a workload with N threads will finalize when the total number of executed instructions is N times the threshold. Typical values for this threshold range from 100-million instructions [23] and 200-million instructions [12] to 1-billion instructions [10]. However, the Fixed Instructions methodology is also unable to ensure that a representative part of every benchmark is being executed, since workload simulation ends in an arbitrary point (whenever the total number of executed instructions is reached). Even worse, despite the total number of instructions is the same, the mix of executed instructions may change. As an example, imagine that two different instruction fetch policies must be compared, IF1 and IF2 in a 2-context SMT processor. IF1 always prioritizes instructions belonging to the first context and IF2 always prioritizes instructions belonging to the second one. The simulation finishes when N instructions from both threads are executed. When both simulations end, they have executed the same number of instructions but the instruction mix is not the same: most instructions belong to the first thread for IF1 and most instructions belong to the second thread for IF2. Therefore, since the executed instructions are not the same, the comparison between IF1 and IF2 is not fair regardless of the metric used.

3.2 Analysis of Current Methodologies

To show the behavior of current methodologies, we analyze three of the most currently used methodologies for evaluating the performance of multithreaded processors: First (F), Last (L), and Fixed Instructions (I). We analyze three versions of the latter: 200-million fixed instructions (I2), 400-million fixed instructions (I4), and 800-million fixed instructions (I8). As an example, Figure 2 shows the obtained results for these methodologies using our SMT simulator configuration and a 2-thread workload composed by the benchmarks *perlbmk* and *gap*. The simulation ends when both traces have executed at least twice.

We provide data for two different fetch policies: the icount policy [21] in Figure 2(a) and the stall policy [20] in Figure 2(b). In Figure 2, the y-axis shows processor performance (IPC) and the x-axis represents execution time. The light-gray bars show the instant IPC of *gap*. Likewise, the dark-gray bars show the instant IPC of *perlbmk* [3]. In every cycle, the sum of both bars represents the instant throughput, *i.e*, the sum of the instant IPC of both threads. The black horizontal line represents the average instant throughput until a time instant, that is, the average value of the instant throughput for every cycle from the beginning of the workload execution until the current time instant. The white circles over the black line show the final throughput reported by every

[3]To obtain the instant IPC of benchs we sample periods of 15K cycles

Table 3. Behavior of current methodologies.

Methodology →		I2	I4	F	L	I8
IPC Throughput $IPC_{gap} + IPC_{perl}$	icount	3.2	3.5	3.5	2.4	2.6
	stall	3.7	4.0	4.1	3.4	3.9
stall Improvement(%)→		13.1	15.1	18.2	41.8	53.0

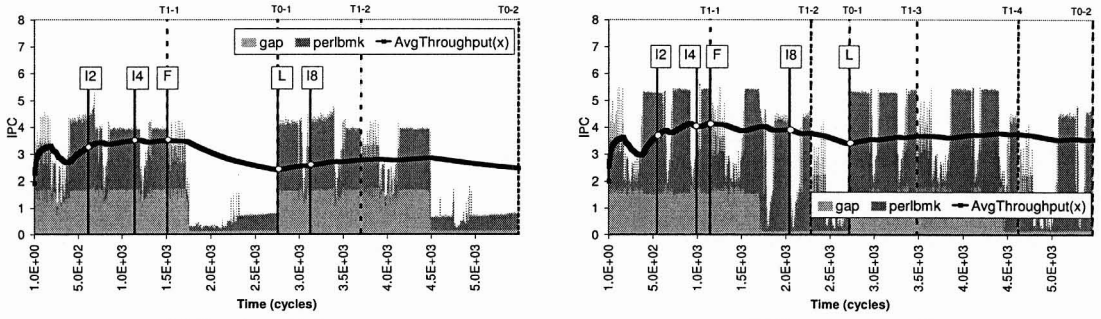
(a) Improvement of stall over icount with the different methodologies.

	Th.	Methodology				
		I2	I4	F	L	I8
Number of full executions	T0	0	0	0	1	1
	T1	0	0	1	1	1
% of instructions (current execution)	T0	26	61	82	0	60
	T1	36	75	0	63	77

(b) # of full executions and percentage of instructions executed of the current execution

methodology and the vertical solid lines show the cycle in which the workload simulation ends according to each experimental methodology. Finally, the vertical dashed lines show the time instant at which every instance of a trace finishes. Above each line we add a legend in the form $Tx - y$, in which x indicates the trace and y indicates the number of times trace x has been executed. The main observation that can be drawn from Figure 2 is that every methodology provides different throughput values. It is summarized in the second (icount) and third (stall) rows of Table 3(a). It should be taken into account that researchers use simulation to evaluate the performance of a design enhancement relative to a baseline design. In the experiment of Figure 2, we can measure the improvement of stall with icount as baseline (shown in the last row of Table 3(a)). Although stall improves the performance of icount for all methodologies, the speedup varies depending on the methodology used. If the $I2$ methodology is used, stall only achieves 13% performance improvement. But if measurements are taken using the $I8$ methodology, stall improvement arises to 53%. That is, depending on the evaluation methodology the stall improvement over icount varies up to 40%. Such a wide range of variation makes difficult to estimate the impact of any proposal and may cause misleading conclusions when a multithreaded processor enhancement is evaluated.

As discussed in previous sections, this problem is due to the fact that current methodologies cannot ensure fully representativity of every trace of the workload, which can lead to unfair comparisons between different simulator setups. Table 3(b) summarizes these drawbacks by showing the number of times every trace has been completely executed and the percentage of instructions executed in the last repetition for each methodology when using the stall fetch policy (results for icount are similar). The total amount of executed instructions varies from one evaluation methodology to another one. For example, in the case of the $I8$ methodology, T0 executes once completely and then executes 60% instructions from a second repetition. The same happens with T1, but in this case the percentage of instructions executed in the



(a) Results with icount (b) Results with stall

Figure 2. IPC of *gap* and *perlbmk* when executed together on the SMT simulator.

second repetition is 77%. Another example is the L methodology: T0 executes once and T1 execute once and 63% of the second repetition. This data clearly shows that the mix of instructions is different in every case what could make the comparison of results misleading.

These representativity and fairness issues are also present in real multithreaded processors [3][14][18]. We made a similar experiment on our real processor environment. We measure the performance throughput of the *gcc* and *gap* benchmarks when they are executed together on a Pentium 4 processor. The conclusions were the same: the real throughput value varies depending on the used methodology. These results are not shown due to space constraints.

4 The FAME Methodology

Current simulation methodologies do not ensure that all traces in a workload are faithfully represented in the simulation results. To alleviate this problem, we propose a new methodology called FAME. The main objective of FAME is to obtain representative measurements of the actual processor behavior.

4.1 Trace Reexecution

In doing so FAME determines how many times a trace in a workload should be reexecuted for being faithfully represented. In order to determine it, FAME analyzes the behavior of every trace in isolation. In this paper we assume that the behavior of each thread in a workload executed in multi-thread mode remains similar to the behavior in single-thread mode since the code signatures do not change. We check this assumption in a aggressive configuration: an out-of-order SMT processor with many shared resources where the interaction between threads in very high. Thus, this configuration represents an unfavorable scenario where we evaluate our assumption. It is clear that, if those proposals work in this hard configuration, they will work better in narrower processors with fewer shared resources.

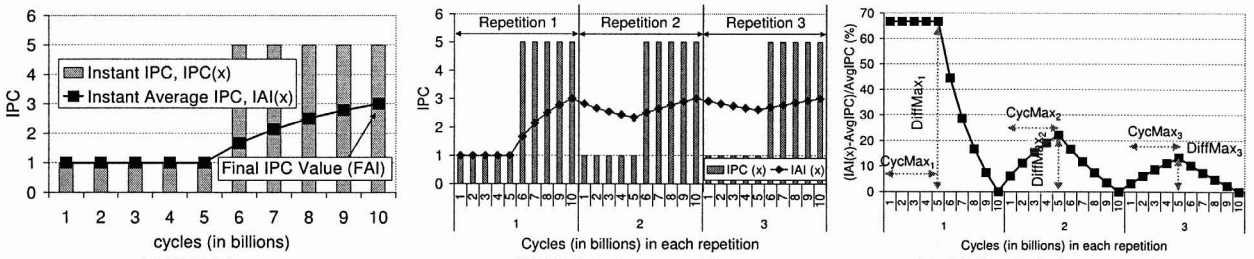
Depending on the particular methodology features, the execution of each thread in a workload may be stopped at any point, and the IPC value provided by the methodology will be the average IPC value until that point. This average IPC would be fully representative of the thread execution if it is similar to the final IPC value, that is, the average IPC value at the end of the whole thread execution. Therefore, the FAME methodology forces each trace to be executed enough times so that the difference between the obtained average IPC and the final IPC is below a particular threshold.We have tested in our simulator that flushing the TLB and cache entries, before a program re-execution, has an effect near negligible in our experiment compared to not flushing.

The basis of FAME can be better explained using a synthetic example. Light-grey bars in Figure 3(a) show the instant IPC of our synthetic application, that is, the IPC on each particular cycle of its entire execution when run in isolation. The black line shows the evolution of the average IPC of the application along its execution. The average IPC value for a given execution cycle is calculated as the average value of the instant IPC from the beginning of the trace execution until that particular cycle. Thus, the final IPC would be equal to the average IPC value at the end of trace execution. It is clear that the average IPC converges towards the final IPC value.

Figure 3(b) shows the instant and the average IPC during three reexecutions of the program. In addition, Figure 3(c) shows the difference between the average and the final IPC during the three reexecutions. It is clear that the average IPC converges towards the final IPC value. Even if that difference is a decreasing function, it is important to note that it is not monotone. This means that the difference would be very small in a given cycle, but it may increase again in the subsequent cycles. Therefore, if the goal is to obtain representative measurements, program execution cannot be stopped at any point.

One could think that the solution is to finalize trace execution when a full application repetition has been executed, since the average IPC is always equal to the final IPC at the end of any repetition. However, a multithreaded processor is able to execute more than one program at once. Although simulation can be stopped at the end of a repetition for one of the programs, it is likely that this point is not the end of a repetition for the other programs, and thus the other could be

0-7695-2944-5/07 $25.00



(a) IPC (x) (b) IPC during repetitions (c) Diff (%) during repetitions

Figure 3. Instant IPC, average IPC, and *difference between both* of a synthetic program during 3 repetitions.

not accurately represented. The actual solution comes from the observation that, although the difference between the average and the final IPC does not decrease monotonically, the maximum difference in a reexecution is lower for each new repetition. That is, it is a decreasing monotone function. Thus, if we execute enough repetitions of a trace, the maximum difference will reach a value small enough to consider that the average per-thread IPC is representative of the full benchmark. For this reason, our methodology reexecutes all traces several times, until the difference is upper-bounded by a given threshold.

Figure 3(c) shows the difference between the average and the final IPC as our synthetic trace is reexecuted. The highest difference values are obtained in the first repetition due to the cold-start IPC calculation of the trace. The difference decreases along with the trace execution, reaching zero when the first repetition finishes. The difference is always zero at the end of every program repetition, since the average IPC is always equal to the final IPC at those points. It can be observed in Figure 3(c) that the IPC behavior of the first repetition is not representative of the IPC behavior in following repetitions due to the cold-start effect. For this reason, we discard the first repetition. It can also be observed that the difference between the average and the final IPC presents similar behavior for all repetitions excluding the first one. Indeed, the cycle in which the difference achieves its higher value is always the same for all repetitions.

$$InstMax_i = (TotalInst * (i-2)) + InstMax_2 \quad (1)$$

$$CycleMax_i = (TotalCycle * (i-2)) + CycleMax_2 \quad (2)$$

Let $InstMax_2$ be the instruction in the second repetition that reaches the maximum difference between the average IPC and the final IPC value within that repetition. Let also $CycleMax_2$ be the cycle in which that instruction is executed. Since the instruction and cycle in which the application reaches the maximum difference is always the same for all repetitions from the second one onwards, we can compute the number of instructions and cycles that should be executed to reach $InstMax_i$ and $CycleMax_i$ for every repetition i. This calculation is performed with formulas 1 and 2, in which $TotalInst$ and $TotalCycle$ are the total number of instructions and cycles of the trace on each repetition.

These equations make possible to compute the maximum difference value for any trace repetition beyond the second one without needing to actually execute it. In other words, executing two repetitions is enough to calculate the maximum difference value for any number of additional repetitions, greatly reducing the simulation time required to obtain these values. Thus, the maximum difference value from the beginning of the first repetition, can be calculated using equation 3.

$$DiffMax_i = \left| \frac{InstMax_i}{CycleMax_i} - FinalIPC \right| \quad (3)$$

From equation 3 we can deduce a formula to calculate the minimal number of repetitions required to ensure representativity of a trace. Since it is not possible to achieve perfect representativity, we define a threshold value that indicates the maximum difference between the average IPC and the final IPC that is acceptable to consider that the average IPC value obtained is representative of the full trace execution. We call this threshold the *Maximum Allowable IPC Variance* (MAIV).

In order to obtain representative results, simulations will not finalize until all threads have reached the point where the maximum difference between the average IPC and the final IPC is smaller than a chosen MAIV. From this point onwards, simulation can be stopped at any time. The result shown in Equation 4, states how to calculate the minimal number of repetitions required to fulfill a given MAIV requirement. This result is obtained working out the value of i from equations 1, 2 and 3. [4]

4.2 FAME in simulation scenarios

As with previous methodologies, the first step to apply the FAME methodology is obtaining a representative trace of every benchmark. We have selected the SimPoint tool [16] to generate them. Once traces are obtained, we simulate two repetitions of every trace in isolation. We sample the IPC of the application in order to get the IPC during execution.

Figure 4 shows the instant IPC of *apsi* (a) and *eon* (b). For this experiment we sample the IPC of each benchmark every

[4] It is not expected the reviewer to work out i by hand, instead a mathematical tool can be used for this purpose.

$$i \geq \left\lceil \frac{(CycleMax_2 - 2 * TotalCycles) * (FinalIPC * (1 + MAIV)) - InstrMax_2 + 2 * TotalInst}{TotalCycles * (FinalIPC * (1 + MAIV)) - TotalInst} \right\rceil \quad (4)$$

Table 4. Repetitions required for every SPEC2K bench

Bench. Name	MAIV(%) 20	10	5	2	1
bzip2	1	1	1	3	6
crafty	1	1	1	1	1
eon	1	1	1	1	1
gap	2	3	7	17	34
gcc	2	3	6	16	32
gzip	1	1	2	4	8
mcf	1	1	1	1	1
parser	1	2	3	8	15
perl	1	2	4	10	20
twolf	1	1	1	1	1
vortex	1	1	1	3	6
vpr	1	1	1	1	1

Spec CPU INT

Bench. Name	MAIV(%) 20	10	5	2	1
ammp	1	1	1	1	1
applu	1	1	1	4	7
apsi	2	3	7	17	35
art	1	1	1	1	1
equake	1	1	1	1	2
galgel	2	3	6	15	30
lucas	1	1	1	1	3
mesa	1	1	1	3	6
mgrid	1	1	2	5	10
swim	1	1	2	5	10
wupw.	1	1	1	2	4

Spec CPU FP

(a) in the simulation environment

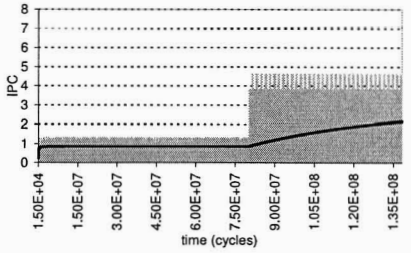
Bench. Name	MAIV(%) 20	10	5	2	1
bzip2	1	1	1	2	3
crafty	1	1	1	1	1
eon	1	1	1	1	1
gap	1	1	1	2	5
gcc	1	1	2	3	7
gzip	1	1	1	1	3
mcf	1	1	2	5	9
parser	1	1	1	1	1
perl.	1	1	3	4	8
twolf	1	1	1	1	1
vortex	1	1	1	1	1
vpr	1	1	2	5	10

Spec CPU INT

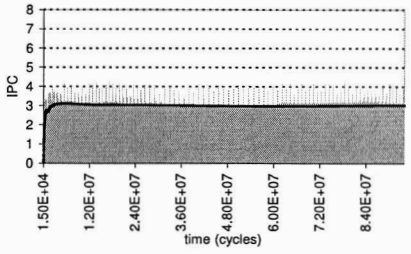
Bench. Name	MAIV(%) 20	10	5	2	1
ammp	1	1	1	2	3
applu	1	1	1	1	1
apsi	1	1	1	1	1
art	1	1	1	1	1
equake	1	1	2	4	7
facerec	1	1	1	1	1
fma3d	1	1	1	1	1
galgel	1	1	1	1	1
lucas	1	1	1	1	1
mesa	1	1	1	1	1
mgrid	1	1	1	1	1
sixtrck	1	1	1	1	1
swim	1	1	1	1	1
wupwise	1	1	1	1	1

Spec CPU FP

(b)in the Intel Pentium 4.



(a) apsi



(b) eon

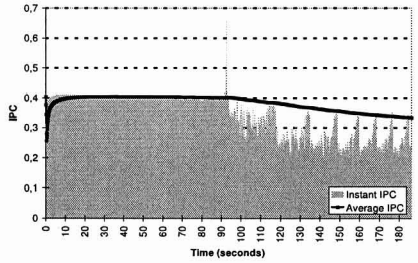
Figure 4. Instant and average IPC of two simulated benchmarks with different behavior.

15,000 cycles. As before, the light-gray bars and the black line represents the instant IPC and the average IPC of the given benchmark respectively. The final IPC is the average IPC at the end of the simulation. Figure 4(a) shows an scenario in which the instant IPC of the application (*apsi*) varies noticeably. On the other hand, Figure 4(b) shows a scenario in which the instant IPC of the application does not vary significantly (*eon*). Intuitively, in order to fulfill a given MAIV, it would be necessary to reexecute more times *apsi* than *eon*, since its average IPC presents more variability. From this information we obtain $CycleMax_2$ and $InstMax_2$, and compute the number of re-executions, i, required to satisfy a given MAIV.

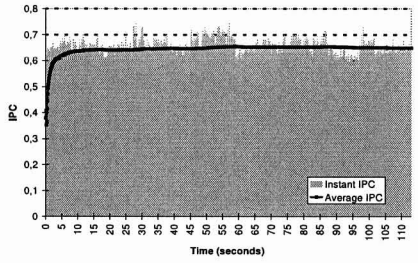
Table 4(a) shows the minimal reexecutions required for both SpecInt and SpecFP with MAIV values ranging from 20% to 1%. The lower the MAIV value is, the higher accuracy required, and thus, usually, the more repetitions are needed. For example, if a MAIV value of less than 1% is required, some benchs (*gap*, *gcc*, *apsi* and *galgel*) have to be reexecuted more than 30 times to be accurately represented in the workload. It is also noticeable that when the MAIV requirements are relaxed (20%) only 1 repetition is needed in most of the SPECs.

Once the traces and the minimal number of repetitions are obtained, workload simulations can begin. Workload simulation will not finalize until every trace in the workload has been executed, at least, as many times as the minimal number of repetitions required for accurate representativity. If any trace reaches this minimal number of repetitions before the rest of the traces, it will reexecute once and again until all traces fulfill their requirements. This is not a problem for representativity, since the maximum difference between the average and the final IPC can only decrease. When all traces have been reexecuted at least the corresponding minimal number of times, workload execution can be stopped at any point, since we can ensure that the results are representative. For example, if the workload composed by *gcc* and *apsi* and a MAIV of 1% is required, *gcc* and *apsi* must be reexecuted at least 32 and 35 times respectively. If *apsi* finishes first, the simulator must reexecute it once and again to keep the complete workload executing, that is, to maintain a fair scenario for the execution of the other thread. Once both benchmarks reach the minimal number of repetitions, simulation finalizes.

It is interesting to note that, when a trace is reexecuted, we flush the data of this thread from the memory hierarchy. This flush procedure is done to prevent the processor from unfairly taking advantage of the warming-up of structures. Indeed, real operating systems do so. In every context switch the TLB is invalidated and thus, the memory hierarchy is flushed. Nevertheless, we found that, for our experimental setup, the initialization part (what could correspond to the first execution instructions after a context switch) is a negligible percentage of the total execution time and it does not vary the results. The difference between flushing and not



(a) vpr



(b) crafty

Figure 5. Instant and avg IPC of two bench in a Pentium 4.

flushing is less than 0.01% for all cases.

4.3 FAME in Real Processors

Our FAME methodology can also be applied to real processor environments, just requiring few changes. The main difference is that benchmarks are executed until completion instead of selecting representative traces, since a real processor executes benchmarks faster speed than a simulator. Besides, benchmark reexecution is done using the operating system support. The memory hierarchy is flushed before each program re-execution because the OS allocates a new process address to execute another instance of the same application. Thus, the thread memory footprint corresponding to the program re-executed is erased by the OS.

As an example, Figure 5(a) shows the instant and average IPC of the benchmark *vpr*, which presents a variable behavior, while Figure 5(b) shows the IPC of the benchmark *crafty*, which presents a nearly constant average IPC (measured every 100 milliseconds). In general, applications executed in a real processor have lower IPC variance, and thus the number of times an application has to be reexecuted in the real processor scenario is usually less than the number of reexecutions needed in the simulation environment. Table 4(b) shows the minimal number of repetitions required per benchmark with MAIV values ranging from 20% to 1% in the real processor environment.

5 Analysis of Evaluation Methodologies

In order to correctly measure the performance of a multithreaded processor it would be desirable that the baseline performance was obtained with the measurements taken when the processor reaches a *steady state* since, in this state, the variation of performance is negligible. In our real processor environment, workloads are composed by full programs. In this scenario, we have measured that the steady state is reached when every program is reexecuted, at least, 20 times in a workload. Following reexecutions do not affect the results. On the other side, in the simulation environment it is not feasible to run full programs on the simulator due to long simulation times. This is the main reason to use representative traces of programs. In this case, given a set of traces, we measured how many times we have to re-execute these traces in a workload to reach an steady state. For the traces used in this paper we measured that the steady state is reached after executing 50 times each trace in a workload. In [24] we show that re-executing single-point traces we obtain similar results (error lower than 4%) than when we re-execute full programs in our simulator.

5.1 Simulation Environment

In a first experiment, we measure per-thread IPC. If per-thread IPC is accurate, our FAME methodology can be used to study any metric, like throughput, weighted speedup or harmonic mean, since per-thread IPC is the only variable parameter used to compute these metrics. In this paper we provide results for the throughput and weighted speedup metrics.We calculate the error of every thread in a workload for every methodology using formula 5, in which $T_iIPC_{steady\_state}$ is the IPC of thread i for the baseline, and $T_iIPC_{method.}$ is the IPC of thread i reported by the methodology under study.

$$ErrorT_i = \frac{T_iIPC_{steady\_state} - T_iIPC_{method.}}{T_iIPC_{steady\_state}}(\%) \quad (5)$$

Figure 6 shows the average error of every methodology respect to the baseline. Data is presented for thread 0, Figure 6(a), and thread 1, Figure 6(b), of every workload. For example, thread 0 in the workload composed by *gap* and *perlbmk* is *gap*, and thread 1 is *perlbmk*. For every methodology, we show the average error (gray bars) and the maximum positive and negative errors. Both figures present different results because we do not simulate any particular workload combination more than once (e.g. if we simulate *gap+gcc*, then we do not simulate *gcc+gap*).

In Figures 6(a) and (b) show that the First and Last methodologies present a significant error. This is due to the fact that when these methodologies finalize the execution of a workload it cannot be ensured that all traces are fairly represented in the final result. For the I_x and FAME methodologies we observe that the more simulated instructions the less the error is. This shows that there is a clear tradeoff between the number of instructions a methodology executes and the error it obtains.

Figure 6(c) shows a detailed analysis of this tradeoff. The x-axis shows, for every methodology, the maximum error

observed for both threads in any of the 276 2-thread workloads of our setup. That is, the maximum error shown in Figures 6(a) and 6(b).

If a methodology leads to the point (x_1, y_1) this means that, on average, x_1 instructions are executed for a maximum error of y_1. Given that the target of any methodology is to achieve the lowest error executing as few instructions as possible, any other methodology leading to a point (x_2, y_2) being $x_2 > x_1$ and $y_2 > y_1$ is worse, since more instructions are executed to obtain a higher error.

We draw two conclusions from Figure 6(c). First, I_{1200} obtains a higher error than I_{1000} and I_{800}, what seems counter intuitive as it executes more instructions than both I_{1000} and I_{800}. However, this is due to the fact that in the I_x methodologies there is no control of the goodness of the finalization point, what can lead to a high maximum errors. Second, we observe that FAME behaves better than I_x methodologies since its executes less instructions to obtain lower error values. For example, FAME with MAIV 20% requires executing 848 millions of instructions on average for an error of 10%. On the other hand, the I_{1000} methodology executes 1 billion of instructions and obtains an error of 19%. Analogousyly, FAME with MAIV 5% executes 1.95 billion instructions on average leading to an error of 5.8%, while the I2000 methodology executes 2 billion instructions obtaining an error of 12%.

The key point here is that FAME adapts the finalization moment of a workload depending on the behavior of the traces that compose that workload. Hence, if a trace presents an invariant IPC FAME executes few instructions for a lower error. For example, when executing the workload $eon + eon$, in which both threads have a plain IPC (see Figure 4(b)), FAME with a MAIV 5% executes only 570 million of instructions and leads to an error of 0.15%. The I_{2000} methodology obtains the same error but executing 4x more instructions. On the other hand, traces with higher IPC variance need to be re-executed several times in order to ensure a fair measurement. For example, in the workload $gap + apsi$, $apsi$ has a high IPC variance (see Figure 4(a)) what makes FAME (MAIV 5%) execute 4.8 billion instructions to obtain an error of 0.6%. The I_{2000} methodology executes 2 billion instructions but leads to an error of 12%.

Hence, FAME provides accurate results per thread while executing fewer instructions than the other methodologies. Moreover, since per-thread performance is the only variable parameter used by most multithreaded performance metrics, FAME is also able to provide accurate results for any of them. For instance, Figure 7(a) shows the global errors of the methodologies taking into account the throughput metric. Again, FAME is the methodology that obtains the lowest errors, ranging from -2% to 2% and from -6% to 7% when the MAIV constrain is relaxed (20%). Note that the error of a methodology is independent of the metric used. Even if some metrics, like weighted speedup, have been proposed to provide *fairness*, the results of these metrics depend on the accuracy of measurements. If those measurements are wrong the results obtained by a metric may be also wrong. As an example, we have measured the error when using the weighted speedup as metric for the 2-thread SMT. The trends are similar to thouse using throughput, Figure 7(a). Measured errors for each methodology are $[-21, 53]$ for First, $[-14, 18]$ for Last, $[-19, 79]$ for I400, $[-14, 19]$ for I800, $[-7, 19]$ for I1000, $[-8, 20]$ for I1200, $[-6, 12]$ for I2000, $[-7, 8]$ for MAIV 20%, $[-5, 7]$ for MAIV 10%, $[-3, 6]$ for MAIV 5%, $[-3, 4]$ for MAIV 2% and $[-2, 2]$ for MAIV 1%.

To show that FAME also alleviates the representativity problems in other scenarios, we test all the methodologies using 4-thread workloads (Figure 7(b)). In this case, only the 6 benchmarks with the highest IPC variability (*gcc*, *parser*, *perlbmk*, *gap*, *galgel* and *apsi*) are used to compose workloads, leading to a total of 126 4-thread workloads[5]. Again, FAME is the methodology that presents the lowest errors, ranging from -14% to 12% when the MAIV constrain is relaxed (20%) and from 1% to -2% when the more accurate 1% MAIV is required. The best results from current methodologies are obtained by Last, which has maximum errors ranging from +12% to -20%.

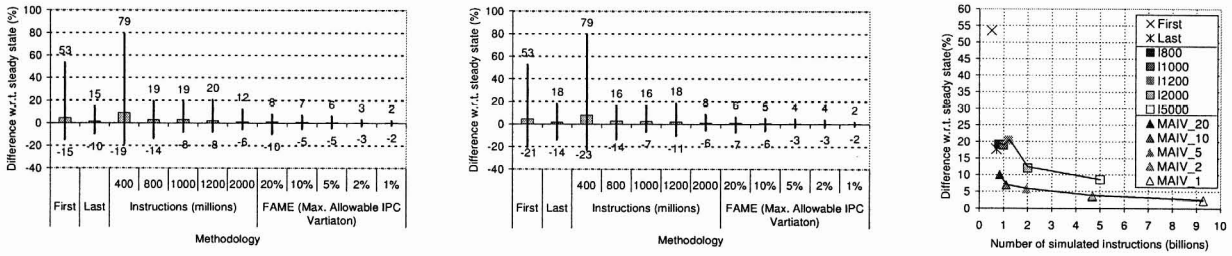
Figures 7(a) and 7(b) also show that, as the number of simulated instructions increases, the accuracy error decreases. For example, in the 2-thread configuration (Figure 7(a)) the 800-million Fixed Instructions methodology presents an error interval from -13% to -16%, whereas the 2-billion Fixed Instructions methodology leads to an error interval ranging from -8% to 6%. It can be observed that, in the 4-thread configuration (Figure 7(b)), the interval of error has increased to -31%, 70% for the 800-million methodology and to -18%, 17% for the 2-billion methodology. The problem with these methodologies is that we cannot fix *a priori* the number of instructions to simulate in order to obtain a low error, since this number depends on both the simulator setup and the number and mix of threads in every workload. In contrast, our FAME methodology presents a much more stable behavior regardless of the configuration, which is a desirable characteristic for any methodology.

5.2 Real Processor Environment

FAME keeps on being the methodology with the lowest error in the real scenario, as shown in Figure 7(c). In this scenario, given that benchmarks are executed until completion, the difference of executed instructions per benchmark is larger than in the simulation scenario, which makes maximum errors become higher. On the other hand, since the time to execute the same number of instructions in the real processor environment is shorter than in the simulation en-

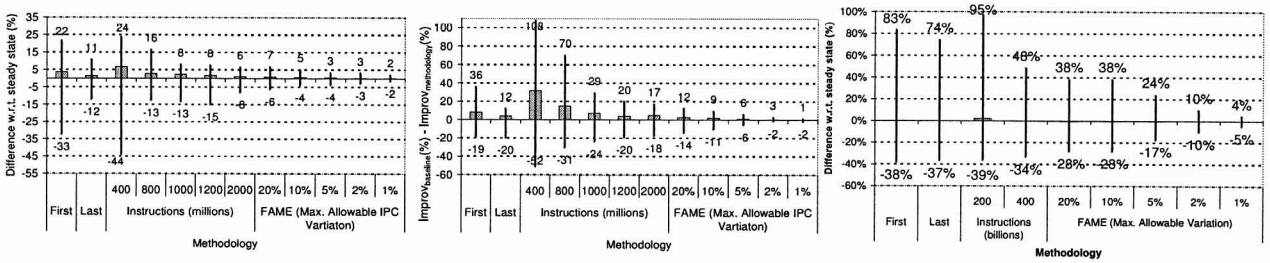
[5]There are 14950 possible 4-thread workload combinations from SPEC2000, making simulation time unaffordable.

0-7695-2944-5/07 $25.00



(a) Thread 0 (b) Thread 1 (c) Error/Instructions tradeoff

Figure 6. Error of the different methodologies for the 2-thread workloads using icount as fetch policy



(a) 2-thread SMT (b) 4-thread SMT (c) Pentium 4

Figure 7. Error in throughput of current methodologies in different scenarios.

vironment, more restrictive MAIVs can be allowed. The errors incurred by FAME are clearly the lowest ones, being the 200-billion instruction methodology the one that presents the worst results (errors range from 95% to -39%). In addition, MAIV 20% executes nearly the same number of instructions that the Last methodology to obtain lower maximun and minimun errors as shown in Figure 7(c). In MAIV 1%, the methodology that obtains the lowest errors (4%, -5%), only 10% more instructions need to be executed compared to the Last methodology.

6 Related Work

Several methodologies and metrics have been proposed for measuring the performance of multithreaded processors executing non-cooperative workloads. On the one hand, evaluation methodologies determine how to take measurements from a workload. In this paper we have evaluated the First, Last and Fixed Instruction methodologies, which have been already explained in previous sections. On the other hand, metrics compute a representative value from the measurements obtained using an evaluation methodology. The most commonly used metrics are throughput [21], harmonic mean [13], and weighted speedup [19].

FAME is an evaluation methodology that provides more accurate measurements than any of the aforementioned methodologies. Like previous evaluation methodologies, FAME is absolutely independent on the technique used to select representative parts of program execution. In particular, the results presented in this paper have been obtained using SimPoint to select a single representative interval per program. Although it is stated in [5] that using a single interval is not accurate enough for multithreaded simulation, we consider it would be not necessarily true. The poor accuracy obtained using a single interval in [5] could be due to lack of representativity of the selected interval, but it could also be due to the fact that this interval is not reexecuted enough times. FAME determines how many times an interval should be reexecuted to provide accurate results and thus it would solve the latter problem.

The co-phase matrix [5] is an evaluation methodology that comprises 3 steps. First, co-phase uses SimPoint to identify program phases and to select a representative interval per phase. Once phases are identified, in a second step a matrix is populated with information for all possible combinations of phases, one cell per each phase combination of traces in the workload, which could be run together during multithreaded execution. Data for each combination of phases is gathered using a few million instructions of detailed simulation. Finally, third step, the multithreaded simulation is done analytically using the contents of the co-phase matrix. The co-phase matrix approach relies on the Fixed Instructions methodology, which we have shown in this paper to be inaccurate. Since the samples that populate the co-phase matrix are generated using the Fixed Instructions methodology, their accuracy cannot be assured. The Fixed Instructions methodology is also used to select the length of the performance estimations. Some differences between FAME and co-phase are the following:

Different architectures: a current drawback of all current methodologies, including FAME, is that any variation in the simulated architecture requires reapplying the method-

ology. In the case of co-phase, after creating a phase-ID trace for each single program, the second and most time-consuming task of the methodology is the co-phase matrix creation. Given that this matrix is populated with IPC values from detailed simulations of combinations of individual phases of each thread, any change in the processor setup likely affect these IPC values, and thus, the matrix has to be recomputed. In the case of FAME, it is required to recompute the number of repetitions for each thread. However, and advantage of FAME is that FAME computes the number of repetitions for each thread in single thread mode. Hence, any change in the architecture that only varies the SMT behavior, and not the particular behavior of a single thread, does not affect FAME. For example, if we change the instruction fetch policy (icount, stall, etc), this variation in the architecture does not vary the IPC of a thread if it is executed alone in the SMT. Therefore, we can use the same number of repetitions across different instruction fetch policies.

Scalability: for each N-thread workload, co-phase builds a matrix of K^N entries, having K as the average number of phases per thread in the workload under consideration. In addition, to populate the matrix it is required to simulate I instructions per entry. Hence, to fill out the matrix, $I \times K^N$ instructions are simulated. In [6], on average, each of the 8 SPEC2000 benchmarks used has, on average, 27 phases ($K = 27$) and each entry of the co-phase matrix is populated with the results of simulating 3.5 million of instructions ($I = 3.5 \times 10^6$). In this scenario, for each 2-thread workload it is required to simulate 2.55 billion (2.55×10^9) instructions, and for each 4-thread workload 1,860 billion (1.86×10^{12}). That is, the size of the co-phase matrix, and hence the instructions to simulate, increase exponentially with the number of threads per workload. On the contrary, the cost of FAME presents a much linear nature because it is applied to separate programs and not to combinations of them. As we have seen in Figure 6(c), the number of instructions executed by FAME depends on the value of the MAIV. For a 2-thread workload, using traces of 300 million of instructions, this number varies from less than 1 billion when MAIV equals 20% to 9 billion when MAIV is 1%. MAIV 5% presents the best trade-off between error and instructions, since it provides a maximum error of 5.8% and executes less than 2 billion instructions. For 4-thread workloads FAME executes from 2,5 billion (MAIV 20%) to 41 billion (MAIV 1%) of instructions. As TLP increases in future processors this scalability problem becomes more accentuated. For a 32-threaded architecture, like the Niagara T1 [3], using traces of 100 million instruction and a MAIV of 5%, FAME executes 10 billion instruction per workload. For this same instruction budget the co-phase approach can only allow 1 phase (Single Point) per program ($3.5 \times 10^6 \times K^{32} = 10 \times 10^9$, so $K = 1.28$), in which case co-phase degenerates in the First methodology.

Real processor environments: opposite to FAME, the co-phase approach cannot be easily applied to measure the performance of real processors. One of the major drawbacks to port co-phase to a real processor scenario is the implementation of the checkpointing mechanism needed to start the execution of one phase of a thread. This means that an operating system should provide a mechanism to restore the whole memory image of a process in a given point. On the other side, co-phase makes a fastforward of 1,5M of instructions per thread in a phase to warm-up memory structures what it is impossible to perform in a real processor. Furthermore, if co-phase is modified to avoid this fastforward, it cannot be ensured that both threads in a phase reach, at the same time, the segment of code to evaluate.

Some authors have realized that in order to get accurate evaluation results for multithreaded architectures it is necessary to take into account the performance variability phenomena [4][5]. In [5], in order to obtain more accurate results, a statically generated co-phase matrix can be used to estimate the performance from different starting points for all threads in a workload [6]: estimations are repeated once and again, using different starting points, until the average result statistically converges for a given level of confidence. The main problem here is that if the estimation for each point is inaccurate it is necessary to increase the number of estimations to converge. FAME, like co-phase [5], provides a simulation methodology to obtain fair measurements for a given starting points for each trace. In this sense, FAME is completely orthogonal to the methodology of [6]. FAME can be used to compute the measurements for each estimation point. We leave the combination of FAME and [6] as future work.

Concerning real processor evaluation, the IBM Power5 (2 cores and 2-threads per core) was evaluated using 4-thread workloads containing the same application replicated four times [18]. Since all the threads in the workload are the same program, they finalize execution almost simultaneously, which means that the error is negligible regardless the evaluation methodology used. However, using just this type of workload limits the variety of the analysis that can be done. FAME would have allowed evaluating the Power5 processor using any arbitrary workload, since it is a more general methodology.

In [22] in a heterogeneous workloads are executed 12 times to guarantee, at least, 3 executions of each program. It is not explained how the number of repetitions are obtained, and since this number depends on both the simulator setup and the number and mix of threads in every workload, this methodology cannot be extrapolate to other environments. The point of FAME is that we fix *a priori* the number of repetitions to simulate in order to obtain a low error.

0-7695-2944-5/07 $25.00

7 Conclusions

To guarantee the resemblance between the real world and the simulation environment in multithreaded architectures is mandatory the use of an appropriate measuring methodology. The evaluation of the capabilities of a multithreaded processor using a given workload requires taking measurements when all the threads in that workload are running. However, the execution speed of every thread in a workload varies according to the particular thread features and the availability of shared resources, which makes some threads finalize execution before others. This fact forces researchers to define, firstly, when the workload execution finalizes and, secondly, when measurements are taken. However, the methodologies currently used to define these features cannot ensure that these results are representative. Even worse, since thread speed also depends on the processor features, any change in the processor setup would vary the mix of executed instructions from every thread, and thus two results obtained using two different processor setups are not comparable.

To deal with these problems we propose FAME, a novel evaluation methodology aimed to fairly measure the performance of multithreaded processors. FAME is mainly based on representative trace reexecution since, when a trace is reexecuted enough times, its average IPC value converges to a representative result. Therefore, once all benchmarks in a workload are executed a required number of times, it is possible to stop workload simulation at any arbitrary point, since representativity is ensured.

As a case study, we apply FAME to a well-known SMT simulation tool and a real SMT processor. In both cases, we have shown that FAME achieves better accuracy than previously proposed methodologies. In addition, any metric can use the measurements obtained with FAME, since a methodology just dictates how to take measurements and not how to use them. Even more, since the main difference among multithreaded designs is the amount of shared resources, all of them present the same evaluation problems, making FAME directly applicable to SMT processors, CMP processors, and even CMP/SMT processors in both simulation and real processor scenarios.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2004-07739-C02-01, and by HiPEAC European Network of Excellence under contract IST-004408. The authors would like to thank Jaume Abella, and Beatriz Otero for their comments.

References

[1] http://www.nas.nasa.gov/software/npb/.

[2] http://www.specbench.org/.

[3] http://opensparc-t1.sunsource.net/

[4] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. *9th HPCA*, 2003.

[5] M. V. Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. *ISPASS*, 2004.

[6] M. V. Biesbrouck, L. Eeckhout, and B. Calder Considering All Starting Points for Simultaneous Multithreading Simulation *ISPASS*, 2006.

[7] D. Burger and T. Austin. The simplescalar tool set, v. 3.0. TR, Computer Sciences Department, University of Wisconsin-Madison, 1999.

[8] F. J .Cazorla, E. Fernández, A. Ramirez, and M. Valero. Dynamically controlled resource allocation in SMT processors. *37th MICRO*, 2004.

[9] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. *In ICCD*, 1996.

[10] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. *36th MICRO*, 2003.

[11] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *Workshop on Workload Characterization*, 2000.

[12] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. *IPDPS*, 2001.

[13] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. *ISPASS*, 2001.

[14] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[15] M. J. Serrano, R. Wood, and M. Nemirovsky A Study of Multistreamed Superscalar Processors. Technical Report #93-05, University of California, Santa Barbara, 1993.

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *10th PACT*, 2001.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatic characterizing large scale program behavior. *10th ASPLOS*, 2002.

[18] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[19] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. *SIGMETRICS*, 2002.

[20] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. *34th MICRO*, 2001.

[21] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd ISCA*, 1996.

[22] N. Tuck and D. M. Tullsen Initial Observations of the Simultaneous Multithreading Pentium 4 Processor *PACT*, 2003

[23] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. *37th MICRO*, 2004.

[24] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernández, and M. Valero. Evaluating Multithreaded Architectures on Simulation Environments Technical Report Universitat Politècnica de Catalunya, UPC-DAC-RR-CAP-2007-16, 2007

[25] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *30th ISCA*, 2003.

[26] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. *Proceedings of CASES*, 2005.